Advanced Dynamic Scripting for Fighting Game AI

Kevin Majchrzak, Jan Quadflieg, Günter Rudolph

▶ **To cite this version:**

Kevin Majchrzak, Jan Quadflieg, Günter Rudolph. Advanced Dynamic Scripting for Fighting Game AI. 14th International Conference on Entertainment Computing (ICEC), Sep 2015, Trondheim, Norway. pp.86-99, ⟨10.1007/978-3-319-24589-8_7⟩. ⟨hal-01758421⟩

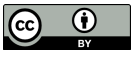
HAL Id: hal-01758421

https://inria.hal.science/hal-01758421v1

Submitted on 4 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Advanced Dynamic Scripting for Fighting Game AI

Kevin Majchrzak, Jan Quadflieg, Günter Rudolph

Chair of Algorithm Engineering, TU Dortmund, 44221 Dortmund, Germany
{kevin.majchrzak, jan.quadflieg, guenter.rudolph}@tu-dortmund.de

Abstract. We present an advanced version of dynamic scripting, which we apply to an agent created for the Fighting Game AI Competition. In contrast to the original method, our new approach is able to successfully adapt an agent's behavior in real-time scenarios. Based on a set of rules created with expert knowledge, a script containing a subset of these rules is created online to control our agent. Our method uses reinforcement learning to learn which rules to include in the script and how to arrange them. Results show that the algorithm successfully adapts the agent's behavior in tests against three other agents, allowing our agent to win most evaluations in our tests and the CIG 2014 competition.

Keywords: Artificial Intelligence · AI · Computer Game · Fighting Game · Dynamic Scripting · Code Monkey · Real-Time · Adaptive · Reinforcement Learning

1 Introduction

Scripting is one of the most widely used techniques for AI in commercial video games due to its many advantages [14]. One of the major downsides on scripted game AI, though, is its lack of creativity. An agent controlled by a classic script may show foolish behavior in any situation the developer has not foreseen and it is an easy prey to counter strategies because it cannot adapt.

Dynamic scripting [14] minimizes the downsides of scripting while retaining its strengths. The idea is to build a static rulebase beforehand and to select and order a subset of its rules to generate scripts on the fly. Commonly reinforcement learning is used for the selection and ordering process whereas expert knowledge is used to design rules. In many cases dynamic scripting is capable of adapting to an enemy's strategy after just a few fights [14]. These results are encouraging but the method is still too slow if enemies change their strategies frequently. Therefore, agents that are controlled by dynamic scripting in its basic form do not perform well against human players or other agents with dynamic strategies.

In the present article we introduce an improved method that meets real-time requirements and, thus, resolves dynamic scripting's shortcoming. The rest of the paper is structured as follows: We give a short introduction to the framework FightingICE used here and discuss related work in section 2. Our advanced version of dynamic scripting is presented in detail in section 3. We compare our solution with the state of the art in section 4 and close with a summary and conclusions.

2 Background

We first introduce the framework used in the Fighting Game AI Competition to make the reader familiar with the most important aspects. Please refer to the official website [4] for details not covered here. We then present related work.

2.1 FightingICE Framework

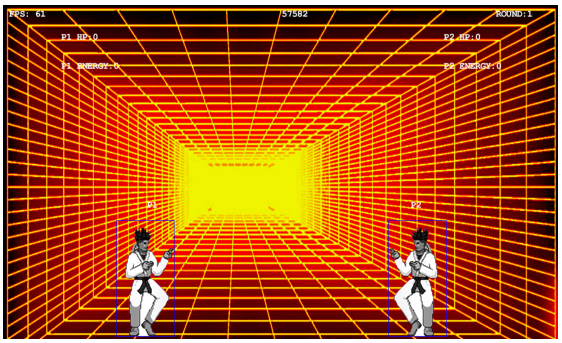


Fig. 1. Screenshot of the FightingICE framework, taken at the beginning of a fight.

The FightingICE framework, used here in version 1.01, is an open source *Beat 'Em Up* game written in Java. Developed and maintained by the Intelligent Computer Entertainment Lab. of Ritsumeikan University, Japan [4, 7] it is the official software used for the Fighting Game AI Competition (FTGAIC), organized by the same group from Ritsumeikan University. The goal of the FTGAIC is to create an agent which controls one of the two characters in the game. The FTGAIC was one of the competitions held at the IEEE Conference on Computational Intelligence and Games 2014 (CIG 2014) [6].

FightingICE implements the classic concept of a Beat 'Em Up game: Two opponent characters fight each other in an arena until a winning criterion is fulfilled. Figure 1 shows a screen capture of the game taken at the beginning of a fight. As shown there, the initial state of the game has the two opponents standing at fixed positions in a neutral posture. Damage and energy values are set to zero. Both characters can move along the x- and y-axis on a two-dimensional grid by using predefined actions. In the context of the FTGAIC a round lasts 60 seconds, one second consisting of 60 frames, which equals 3600 discrete time steps per round. An agent controlling a character has to react in real-time, which means it has $1/60 \approx 0.017$ seconds to decide which action to take. The last action is reused by the framework if an agent violates this constraint.

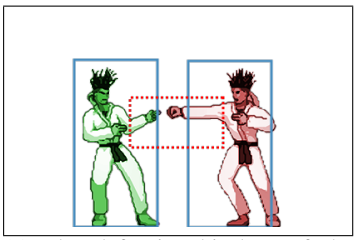
At the end of a round, a total of 1000 points are split between the two characters based upon the damage one inflicted to the other:

$$\mathcal{P} := \begin{cases} \dfrac{\overline{\mathcal{H}}}{\mathcal{H} + \overline{\mathcal{H}}} * 1000 & \text{iff } (\mathcal{H} + \overline{\mathcal{H}}) \neq 0 \\ 500 & \text{otherwise} \end{cases} \tag{1}$$

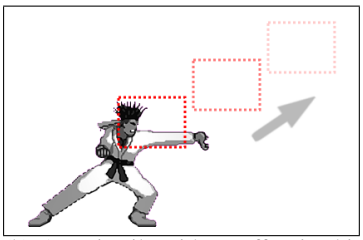
where $\mathcal{H}$ and $\overline{\mathcal{H}}$ are the damage values of the two characters. A fight consists of three rounds. The winner is the agent that gained the most points during the three rounds, which means that 1501 points are sufficient to win.

What makes FightingICE interesting is an artificial lag in the data provided to an agent. At time step t, an agent receives the game state belonging to the time step 15 frames ago. This simulates a response time of 0.25 seconds, similar to a human player's. During the first 15 frames of a round, an agent receives the initial game state.

Agents can perform five different kinds of actions. *Basic actions* are the neutral posture in three variations: standing, crouching and in the air. *Movement actions* are used to move the character around. *Guard actions* can block or weaken the damage of an attack by the opponent. *Recovery actions* are automatically performed if the character has been hit by the opponent or landed after a jump. An agent cannot control the character during a recovery action. *Skill actions* are used to attack the opponent. A skill action consists of three distinct phases: startup (lasts 5 to 35 frames), active (2 to 30 frames) and recovery (8 to 58 frames). Damage is dealt to the opponent only during the active phase and only if the opponent is hit. During the other two phases, the character cannot be controlled and is particularly vulnerable to attacks of the opponent. Detailed information on the 56 different actions can be downloaded from the competition website [4].



(a) The defensive hit box of the left character (left rectangle) collides with the offensive hit box of the right character (dashed rectangle in the middle).



(b) A projectile with an offensive hit box (dashed rectangle) moves away from the attacking character with a constant velocity vector (grey arrow).

Fig. 2. Examples for hit boxes and collisions.

Whether or not a character receives damage from an attack is determined by offensive and defensive hit boxes, modeled as axis aligned bounding boxes (see figure 2(a)). The hit boxes approximate the outline of the character or of parts of the character's body. The size and position of the hit boxes depend on the current actions performed. A character is hit if the intersection of its defensive hit box and the offensive hit box of the attacker is not empty. In this case, the victim receives damage based upon the current actions of both characters. Few skill actions create moving offensive hit boxes which move through the arena, independent from the character (see figure 2(b)). They simulate magical powers, e.g. fireballs.

Some skill actions, if successful, increase the energy level of the character based upon the damage inflicted on the opponent. The maximum energy level is 1000 points. Other skill actions consume energy, which means that they can only be performed if the character has enough energy and that the energy level is reduced by a certain amount if such an action is performed. In other words, the energy can be regarded as a reward for successful attacks and this reward can then be spent to perform other, usually very powerful, actions.

The information available to an agent is incomplete, due to the artificial lag and, to our knowledge, no universal dominating strategy exists, due to the diversity of possible attacks and counter actions. A successful agent therefore needs the ability to act under incomplete knowledge and to adapt its behavior to opponents with new strategies in real-time. Both aspects make the FTGAIC an interesting testbed for AI and learning methods.

The FightingICE framework showed nondeterministic behavior during our experiments. We believe that this noise is introduced by synchronization issues in the parallel part of the code. This adds a further obstacle an agent has to deal with. During our experiments, we did our best to minimize the influence of other processes running on the same machine, to keep the conditions as homogeneous as possible.

2.2 Related Work

In the context of this paper, the most important contribution from literature is the work by Spronck et al., *Adaptive game AI with dynamic scripting* [14], which we use as a basis for our approach. Spronck et al. use a mixture of a classic rulebase (scripting) and reinforcement learning (RL), which they call *dynamic scripting*. They apply their approach to a group of agents for a simulated and a commercial computer role-playing game. As the agents are of different types like *warrior* or *wizard*, which are capable of using very different actions, one rulebase for each type is designed manually. A learning method based on reinforcement learning selects subsets of these rulebases to create one script per agent on the fly. Spronck et al. test their group of agents in combats against other groups with similar basic abilities but static strategies. The scripts are evaluated and modified only once after each combat and it usually takes the learning method several combats to successfully adapt to an opponent. Thus, dynamic scripting in its basic form does not adapt the agents' strategies in real-time.

The primary order of rules within scripts is fixed and based on manually assigned integer values called priorities. Rules with equal priority, though, are ordered by the learning algorithm itself. Further work by Timuri, Spronck and van den Herik [18] demonstrates that the order of rules within scripts can be learned, leading to solutions of equal or better quality and a convergence rate only slightly slower compared to Spronck's original approach. Other articles on dynamic scripting suggest to generate new rules or even complete rulebases automatically [16, 9, 11, 15]. The presented ideas and results are encouraging.

There are a number of publications that are directly related to FightingICE. The software itself is described in the article *Fighting Game Artificial Intelligence Competition Platform* by Lu et al. [7]. Up to the FTGAIC in the year 2013, all agents submitted to the competition were based on static rule bases or finite state machines. For the first

time, adaptive agents like ours were submitted to the FTGAIC 2014. The source code and short descriptions of all entries can be found on the competition website [4].

The organizers of the competition themselves provide an agent called *Mizuno AI* which is based on k-nearest neighbor clustering and is able to adapt its behavior in real-time [19]. The clustering is used to predict the next action of the opponent. Mizuno AI then simulates the outcome of various counter actions and chooses the one with the best possible outcome. The agent proved to be competitive when compared to the best three entries of the 2013 competition.

Another agent that has been developed for FightingICE is described in the article of Park et al. [10]. This agent searches for similar situations in massive play data to decide on promising actions. The data has been collected and stored during training fights against the FTGAIC 2013 participants and two sample bots. The developed agent showed good results in tests against a randomly acting sample bot and Mizuno AI but it was clearly outperformed by the FTGAIC 2013 winner called T.

A different approach for Fighting Game AI commonly used is the imitation of a human player's behavior [17, 13]. The article of Lueangrueangroj and Kotrajaras [8] presents an improved version of Thunputtarakul's *Ghost AI* [17], which imitates a human player's actions after an offline learning phase. The agent of Lueangrueangroj et al., however, is able to mimic its opponent's behavior in real-time. Furthermore, it evaluates the performance of learned actions and uses effective rules more frequently. As a consequence, the improved agent is able to partly adapt its actions to its opponent's strategy in real-time.

Ricciardi and Thill reduce fighting games to plain old Rock-Paper-Scissors [12]. They argue that the central aspect of such a game is to predict the action of the opponent and to react with an appropriate counter action. This problem is therefore modeled as a Markov decision problem and solved with a RL algorithm. For the simple case of only three available actions they created an agent that is able to adapt its behavior to an opponent in real-time. But as soon as they expanded the state space by adding four more actions, the RL algorithm failed to adapt the behavior sufficiently fast. This approach is therefore not applicable to a real-time game like the FightingICE framework.

A classic approach, which we found in many articles, was the use of a combination of reinforcement learning and neural networks [1, 5, 3]. The techniques, applied in these articles, work well for very simple games but need long training phases and are, therefore, not appropriate for real-time games. Cho, Park and Yang compare the value of genetic algorithms, neural networks and evolutionary neural networks for fighting game AI [2]. They come to the conclusion that evolutionary neural networks are the most appropriate technique among them, due to its convergence speed and ratio. But even evolutionary neural networks needed many thousand iterations to adapt to a simple opponent in their tests.

3 Adaptive Generation of Scripts in Real-Time

This section presents an improved version of Spronck's dynamic scripting method. First, we introduce an alternative and more powerful definition for rules and scripts and

give some brief suggestions on rulebase design. Then we explain our learning method and apply it to an agent created for the Fighting Game AI Competition.

3.1 Rules and Scripts

An agent for FightingICE has to consider many variables, including the characters' positions, speed vectors, current actions and past actions, in real-time to decide on appropriate measures. Furthermore, it needs to take into account the simulated delay and stochastic state changes of the game. To cope with the large and high-dimensional search space we will develop a set of rules, based on expert knowledge. These rules classify and, thus, reduce the search space.

Definition 1. *A **rule** R is a mapping $R\colon Z \mapsto \mathbb{B} \times A$. Where $\mathbb{B} = \{0, 1\}$, Z the set of possible game states and A the set of possible actions. We say that $z \in Z$ **fulfills** R iff $R(z) = (1, a)$ for some $a \in A$.*

Definition 2. ***E(R)*** $:= \{z \in Z \mid z \text{ fulfills } R\}$ *is the **fulfilling set** of R. We call R an **empty rule**, iff $E(R) = \emptyset$ and a **default rule** iff $E(R) = Z$.*

The definition of rules in this article differs a lot from the usual understanding of the term *rule* and even Spronck's definition [14]. A rule in our context represents a sub-agent, which maps the entire state space of the game to a set of actions. As a consequence, one single rule could possibly control all of the agent's actions. This special case would be equivalent to the classic scripting approach.

The combination of rules, though, is what gives the learning method the ability to generate highly specialized scripts on the fly. In our method, if a rule is fulfilled, it thereby informs the agent that it believes to know a good solution for the current situation. In general, the fulfilling sets of rules are not disjoint and multiple rules could be fulfilled at the same time. The following definition explains how to manage these situations.

Definition 3. *Let $S = \{R_1, \ldots, R_n\}$ $(n \in \mathbb{N})$ be an ordered set of rules and $Prio\colon S \mapsto \mathbb{Z}$. S is called **script** iff*

$$Prio(R_1) \leq \ldots \leq Prio(R_n) \tag{2}$$

and

$$\bigcup_{R \in S} E(R) = Z\,. \tag{3}$$

*We call $Prio(R)$ the **priority** of $R \in S$.*

The execution of a script generates a candidate solution (action) based on the current game state and script. Rules that are part of the script are traversed according to their order until the first fulfilled rule is reached. This rule determines the action that is returned by the script. Equation 3 assures that every game state is part of at least one rule's fulfilling set and, therefore, the script's execution will always yield a valid result.

According to inequality 2, rules in a script are sorted by priority in ascending order. Therefore rules with low priority determine the script's result before rules with high priority. The order of rules with equal priority is not prescribed and can be decided by the learning algorithm itself. For further information on this topic, please see section 3.3.

3.2 Rulebase Design

The design of a good rulebase is a long and iterative process in many cases and it depends highly on the specific use case. For this reason, concepts that we have learned during the development of our rulebase may not hold true for other scenarios. Nevertheless, we want to give the interested reader a brief overview of our rulebase and the underlying principles.

We designed 28 rules and priorities based on expert knowledge (see appendix A). Rules with low priority are mostly counter-strategies, which are fulfilled in very few and specific situations to avoid them from dominating the script's results. Their early position in the script assures that every chance for a counterattack is taken. Many offensive rules share priority -1. Thus the learning method can independently decide which of these attacks is most effective on a specific enemy. Near the end of the script there are rules that handle situations in which their predecessors have failed. They buy time for the agent to change its strategy. To accomplish this the enemy is avoided and held at distance.

Two rules, a default rule called *PSlideDefault* and a very essential defensive rule called *MFBCtr*, are added to the script manually without taking the learning method into consideration. This kind of interference should only take place very rarely as it reduces much of the algorithms' degree of freedom. It can be an effective option in limited cases, though. The default rule receives the highest possible priority to assure that it only takes action if no other rule is fulfilled. Its job is to assure that the agent is able to decide on an action in any possible situation.

Some of the developed rules are able to perform loops of actions that, if successful, dominate and damage the opponent very effectively. We observed that in some situations these loops were canceled too soon even though they performed really well. To improve our agent's performance we decided to implement a mechanism that informs other rules on successfully running loops. The other rules will take this information into account when deciding whether they are fulfilled or not.

The quality of the agent depends highly on the script's maximum length which is set to 20 within this work. On one hand, if the limit is chosen too low the generated scripts do not reach the needed complexity to produce decent behavior. On the other hand, if scripts grow too long the agent's behavior will be dominated by rules with low priority because rules at the end of the script might never be reached. A good length depends very much on the specific game and rulebase design. 27 of the 28 developed rules are added to the rulebase exactly once. The remaining rule is an empty rule and it is added to the rulebase 17 times. This allows the learning method to vary the script's effective length between 3 and 20 rules. The rulebase contains an overall number of $27 + 17 = 44$ rules.

3.3 Learning Method

To generate scripts in real-time we need a learning method that is able to handle dynamic objective functions and to learn without examples. Furthermore, the method should be able to handle the delayed response and stochastic state changes of the game.

The problem's nature allows for a very straightforward reinforcement learning algorithm that fulfills all of the mentioned constraints. The players' points and their gradients are closely connected to the agent's performance. This significantly simplifies the search for a decent evaluation function. Furthermore, reinforcement learning uses parameters that are readable and understandable for humans. Due to the presented advantages we choose reinforcement learning as the base for our learning method. Moreover, our choice allows us to benefit from the theory and results of Spronck et al. as they have also used reinforcement learning for script generation. Nevertheless, other learning methods could be considered in future research as well.

In fighting games it is advantageous to evaluate sequences of time steps rather than just single actions. This strongly reduces the impact of random noise and smooths the evaluation function. Furthermore it allows the learning method to rate the overall script's performance rather than just the value of single actions on their own.

Definition 4. *The evaluation function* $\mathcal{F}_{a,b} \colon \mathbb{N} \times \mathbb{N} \mapsto [0,1] \subset \mathbb{R}$ *is given by*

$$\mathcal{F}_{a,b} := \begin{cases} \dfrac{\overline{\mathcal{H}}_{ab}}{\mathcal{H}_{ab} + \overline{\mathcal{H}}_{ab}} & \textit{iff } (\mathcal{H}_{ab} + \overline{\mathcal{H}}_{ab}) \neq 0 \\ 0.5 & \textit{else} \end{cases}$$

where $\mathcal{H}_{ab}$ *(*$\overline{\mathcal{H}_{ab}}$*) is the damage that the player's (opponent's) character received during the time steps* $a, \ldots, b$*.*

The strategy for rule selection is controlled by weights $w \in [W_{min}, W_{max}] \subset \mathbb{N}$. Each rule in the rulebase is associated with exactly one weight. The higher a rule's weight, the likelier it will become part of the agent's script. The algorithm used for rule selection draws one rule at a time from the rulebase without replacement. The chance for a rule with weight w to be drawn is approximately w/sum, where sum is the weight of all rules that are not part of the script yet. Once a rule has been drawn, it is inserted into the script according to its priority. Rules that have an equal priority are sorted by weight. The higher a rule's weight, the earlier its position in the script. If the rules' weights are also equal, their order is random.

At the start of each fight every rule receives the same initial weight. Based on the evaluation function a reward is calculated, which the agent aims to maximize by adapting the rules' weights accordingly. In this work the weights are adjusted and a new script is generated every 4 seconds by the learning algorithm. The longer these periods of time are chosen, the longer it takes the agent to adapt to enemy behavior. But if they are chosen too short the agent's decision making will be effected by random noise. Experiments indicated that 4 seconds seem to be a good trade-off in our case.

The reward $\triangle w$ for the time period of evaluation $[a, b]$ is calculated via

$$\triangle w = \begin{cases} -\left\lfloor P_{max} \dfrac{B - \mathcal{F}_{a,b}}{B} \right\rfloor & \text{iff } F_{a,b} < B \\ \left\lfloor R_{max} \dfrac{\mathcal{F}_{a,b} - B}{B} \right\rfloor & \text{otherwise} \end{cases} \tag{4}$$

where P_{max} and R_{max} are the absolute values of the maximum penalty and reinforcement. B is the value of the evaluation function $\mathcal{F}_{a,b}$ for which the agent is neither

punished nor reinforced. The bigger B gets, the harder it becomes for the agent to receive reinforcement and the more frequently it will get punished. For smaller values of B the opposite is the case.

The new weight of rules with weight w that have fired (determined the result of the script) during the time period of evaluation is $w+\triangle w$. Rules that have not fired but been part of the script receive a fifth of the reward and, therefore, the new weight $w + \frac{1}{5}\triangle w$. The weights are clipped to W_{min} or W_{max} if they exit the interval $[W_{min}, W_{max}]$. The factor of $\frac{1}{5}$ is a result of trial and error and could possibly be further optimized. Our first tests were made with a factor of $\frac{1}{2}$ as suggested by Spronck et al. in their paper on dynamic scripting [14]. This value seems to be much too high for frequent weight updates, though. The weights of rules that fire rarely would reflect only the overall script's performance and neglect their individual quality. The much lower value of $\frac{1}{5}$ reduces this effect without ignoring the overall script's performance in our case.

Weights of rules, which are currently not part of the script, are adapted to assure that the the overall sum of weights in the rulebase stays constant. They all receive the same positive or negative adjustment. As a consequence their weights go down if the script generates good results and up if the script performs badly. Therefore, scripts that perform well will not be modified frequently. But as soon as the script's performance drops, even rules with previously very low weight will receive another chance to become part of the script. This is one of the major advantages of dynamic scripting.

In this work we chose $W_{min} = 0$, $W_{max} = 200$, $P_{max} = 80$, $R_{max} = 80$, $B = 0.8$ and the initial weight to be 80. In the following discussion we assume that only one of these values may be varied at the same time while the others remain constant. This assumption is necessary because the parameters influence each other mutually. Because of $W_{min} = 0$ the learning method is allowed to reduce the chances of bad performing rules of being part of the script to zero. As previously explained this does not prevent these rules from regaining weight as soon as the script performs badly. W_{max} controls the agent's variability. Spronck et al. chose the very high value of 2000 for this parameter in their article [14]. This allows the rules' weights to grow effectively unbounded. Thus, the agent's strategy will eventually become very static, once it has adapted to the opponent. In our work W_{max} is set to a much lower value because our agent will have to adapt to non-static enemies as well.

The parameter B is set to the very high value 0.8. As a result neutral and random behavior will not be reinforced and the agent will begin to adapt its behavior long before it starts to perform badly. This enables the agent to switch smoothly from one strategy to another as soon as the requirements change. If the agent does not perform well, the combination of high values for P_{max}, R_{max} and B results in rapid reorganization of the agent's script. This speed is crucial for the generation of counter-strategies in real-time and on the fly. An undesired side effect of the strict reinforcement and punishment is that the learning algorithm may discard good strategies prematurely. Then again, the nature of dynamic scripting will assure that falsely discarded strategies will eventually return if the script performs poorly without them. This process is even accelerated by the short update periods of 4 seconds. Therefore the high value of B does not only discard rules rashly, but it also assures that they get another chance very soon. In consequence, this side effect is not a major downside of our method.

If the rule's initial weights are chosen too low compared to W_{max}, it is likely that a very small number of rules will share all the available weight between them, while the other rules' weights might drop near zero. This would limit the agent's complexity and is, therefore, undesirable. On the other side, if the initial weights are chosen too high compared to W_{max}, this may lead to a lot of rules with high weights. As a consequence, the selection process will become very random. The value of 80 for the initial weights is a compromise between these contrary effects, which works well in practice for our agent.

4 Results and Discussion

In this section we will evaluate our agent's performance in fights against three other agents called *T*, *Mizuno AI* and *Airpunch AI*. To distinguish our agent from the others we name it *Code Monkey (Version 1.1)*. Code Monkey has been tested in 100 fights against each opponent with an overall number of 900 rounds and a running time of approximately 15 hours. The number of repetitions does not influence our agent's performance because it is reinitialized with no prior knowledge about its opponent before every fight. The repetitions' purpose was merely to reduce random noise in our results.

The winner of the FTGAIC 2013 called T is based on a static rulebase. The agent's source code can be accessed on the contest's homepage [4]. T prefers executing slide tackles, but they are also its own greatest weakness. Mizuno AI has already been described in section 2.2. Its ability to adapt to enemy behavior in real-time makes it a highly interesting opponent. Here, Code Monkey needs to adapt its strategy faster than Mizuno AI to succeed. The last opponent used for evaluation is AirPunch AI. This agent has been developed by our team to test Code Monkey's ability to counter air attacks. AirPunch AI jumps up in the air repeatedly to avoid being hit and then it attacks with a punch diagonally towards the ground. The illustrated combination can be repeated rapidly and a hit inflicts a high amount of damage. Because of the framework's simulated delay an enemy that uses this combination is very hard to control. It showed that AirPunch AI performs outstandingly well against most enemies what makes it even more interesting as an opponent for our agent. Airpunch AI is vulnerable to attacks that hit the enemy at a distance and in midair. The most important rule of our agent in this context is called PDiagonalFB.

Figure 3 shows the frequency of usage for Code Monkey's rules during our tests in percent. Firstly, the percentages for each fight had been calculated and then they where arithmetically averaged. For clarity, only frequently used rules are explicitly named in the figures. Percentages of the remaining rules are summed up as *Others*. More than 80% of Code Monkey's behavior against each enemy was determined by the top three to five rules. This is a major specialization since the rulebase contains 44 rules to choose from. It turns out that the rules most frequently used against T and AirPunch AI aim right at their weaknesses (please see figure 3 and the rules explained in appendix A).

Figure 4(a) shows box plots of the results (Code Monkey's points) after every round and fight. At the end of each round both agents split 1000 points between them based on the damage they received. A fight consists of three rounds and is won if the sum of the agent's points exceeds 1500. Code Monkey dominated Mizuno AI and has won

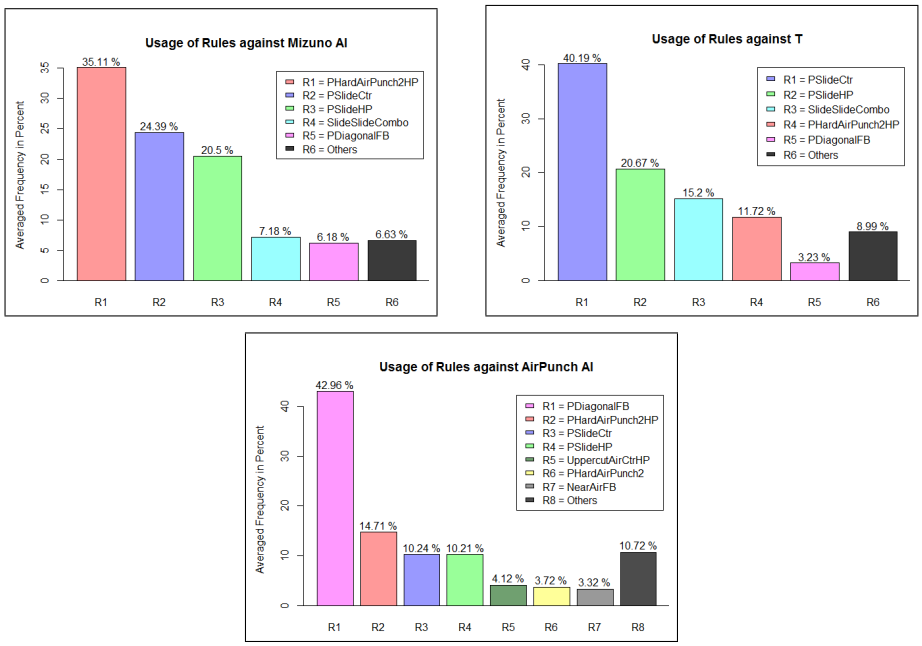
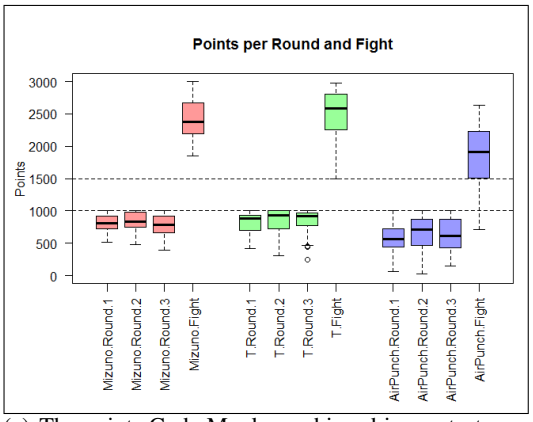


Fig. 3. The rules' averaged frequency of usage during the fights against T, Mizuno AI and AirPunch AI.

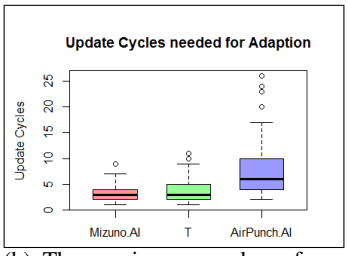
all of the 100 fights. On average, Code Monkey gathered even more points against T and won 99 fights (one loss). It is fair to say that T, although being the winner of the 2013 competition, is no match against Code Monkey. The picture is not that clear in the comparison with AirPunch AI: Code Monkey was able to win 75% of the fights but lost the other 25% (no draws). A Wilcoxon signed-rank test with continuity correction confirms that the results are statistically significant: When testing with the null hypothesis that the true median of Code Monkey's points is 1500 and the alternative that the true median is greater, we get a p-value of 1.109×10^{-9}. Furthermore the true median is greater than 1749.5 at a confidence level of 99%. It is, therefore, save to say that Code Monkey is indeed better than AirPunch AI.

When measuring the speed of adaption, it comes down to the question: When do we know that Code Monkey successfully adapted to an enemy? We chose to compare the agent's points after every update period (4 seconds) with the points before this period. If our agent's point value has fallen, this means it received more damage than its opponent and, thus, it is likely that Code Monkey still has not adapted to the enemy well. Moreover, the use of time periods for evaluation minimizes the influence of random noise on the calculated trend.

Figure 4(b) shows the maximum number of update cycles Code Monkey needed to adapt to its enemy for each fight. We used a Wilcoxon signed-rank test with continuity correction to check if the results are statistically significant: When testing with the null hypothesis that the true median of the required update cycles per fight to adapt to



(a) The points Code Monkey achieved in our tests per round and fight.



(b) The maximum number of update cycles Code Monkey needed to adapt to its enemy per fight.

Fig. 4. Achieved points and speed of adaption.

Mizuno AI, T and AirPunch AI is 3.5, 4.5 and 9 and the alternative that the true median is smaller, we get p-values of 4.935×10^{-5}, 2.745×10^{-5} and 7.165×10^{-5}. Furthermore at a confidence level of 99% the true median of the times required to adapt to Mizuno AI, T and AirPunch AI is smaller than 12, 16 and 32 seconds.

These are great results as they imply that Code Monkey is able to adapt rapidly, even to strategies that change in real-time themselves. On average it took Code Monkey twice the time to adapt to AirPunch AI and there were some outliers with even much longer times. Nevertheless, in many cases Code Monkey adapted to AirPunch AI very fast or at least fast enough to win the fight.

5 Conclusions

We presented an enhanced version of dynamic scripting and applied it to an agent for fighting games. Our agent called Code Monkey (Version 1.1) outperformed its opponents in our tests and won the CIG Fighting Game AI Competition 2014. Furthermore, our tests have shown that Code Monkey is able to adapt to static and dynamic strategies in real-time and on the fly in less than 12 to 32 seconds on average. The learning method proved to be very resistant to random noise and capable of handling stochastic state changes. In most cases the agent turned out to be reliable, even though there were a few outliers in our tests. Detailed results of the competition and Code Monkey's commented source code (including the rulebase) can be downloaded from the competition's homepage [4]. Therefore, our results can easily be replicated.

There are many promising opportunities for future work. Some examples, like the automatic ordering or generation of rules, have already been mentioned in section 2.2. Changes to the evaluation function could shift the agent's primary goal from wining to, for example, entertaining the player or behaving like a human player. In any case, there

is a lot of room for creativity on this topic. In our opinion, the approach presented in this article is very relevant for practical game development. Due to the method's close relation to scripting, game developers can reuse their existing scripts and experience and still greatly increase their AI's value.

A Appendix

Table 1. Names, priorities and short descriptions of the rules that control our agent.

Rule	Prio	Description
MFBCtr	-10	Counters strong enemy projectiles (megafireballs).
MFBTimeExtLow	-6	Throws strong projectile (megafireball) if time is very low.
MFBEnemyDownHP	-5	Throws strong projectile if enemy is on the ground.
MFBTimeLow	-5	Throws strong projectile if energy is high and time is low.
ShortSlideCtr	-3	Counter for too short enemy slide tackle.
UppercutAirCtrHP	-3	UppercutAirCtr with higher priority.
ShortAttackCtr	-2	Counter for too short enemy attacks.
SlideCtr	-2	Jump over enemy slide tackle and counterattack from behind.
KneeAirCtrHP	-2	KneeAirCtr with higher priority.
DistanceAirFB	-1	Attack from distance using a projectile if enemy is in the air.
NearAirFB	-1	Projectile from near range if enemy is in the air.
FBCtr	-1	Counter for enemy projectiles.
PSlideHP	-1	Executes a slide tackle if possible.
SlideSlideCombo	-1	Combination of multiple slide tackles.
PSlideCtr	-1	Uses SlideCtr to attack the enemies back.
PHardAirPunch2HP	-1	PHardAirPunch2 with high priority.
PDiagonalFB	-1	Executes an uppercut and throws projectiles at the enemy if possible.
WrongDirectionCtr	0	Attack from behind if enemy is facing the wrong direction.
AirSlide	1	Attack with a punch in midair if the enemy is near.
UppercutAirCtr	1	Counters air attacks with an uppercut.
KneeAirCtr	2	Counters air attacks with a knee strike.
PHardAirPunch2	2	Jump and then a punch (type 1) in midair (if possible).
DistanceFB	3	Attacks the enemy from distance using a projectile.
FleeJump	3	Avoids the enemy by jumping away.
Uppercut	3	Executes an uppercut when the enemy is near.
PHardAirPunch	9	Jump and then a punch (type 2) in midair (if possible).
EmptyRule	20	Rule with an empty fulfilling set.
PSlideDefault	100	Executes a slide tackle if possible (default rule).

References

1. Cho, B.H., Jung, S.H., Seong, Y.R., Oh, H.R.: Exploiting Intelligence in Fighting Action Games Using Neural Networks. IEICE - Trans. Inf. Syst. E89-D(3), 1249–1256 (Mar 2006)
2. Cho, B.H., Park, C., Yang, K.: Comparison of AI Techniques for Fighting Action Games - Genetic Algorithms/Neural Networks/Evolutionary Neural Networks. In: Entertainment Computing - ICEC 2007, Lecture Notes in Computer Science, vol. 4740 (2007)

3. Cho, B., Jung, S., Shim, K.H., Seong, Y., Oh, H.: Reinforcement Learning of Intelligent Characters in Fighting Action Games. In: Entertainment Computing - ICEC 2006, Lecture Notes in Computer Science, vol. 4161, pp. 310–313 (2006)
4. Fighting Game Artificial Intelligence Competition. http://www.ice.ci.ritsumei.ac.jp/~ftgaic (2015), accessed: 2015-02-02
5. Graepel, T., Herbrich, R., Gold, J.: Learning to fight. In: Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education. pp. 193–200 (2004)
6. IEEE Conference on Computational Intelligence and Games 2014. http://www.cig2014.de (2014), accessed: 2015-02-02
7. Lu, F., Yamamoto, K., Nomura, L., Mizuno, S., Lee, Y., Thawonmas, R.: Fighting game artificial intelligence competition platform. In: Consumer Electronics (GCCE), 2013 IEEE 2nd Global Conference on. pp. 320–323 (Oct 2013)
8. Lueangrueangroj, S., Kotrajaras, V.: Real-time imitation based learning for commercial fighting games. In: Computer Games, Multimedia and Allied Technology (CGAT), 2nd Annual International Conference on. pp. 1–3 (2009)
9. Osaka, S., Thawonmas, R., Shibazaki, T.: Investigation of Various Online Adaptation Methods of Computer-Game AI Rulebase in Dynamic Scripting. In: Proceedings of the 1st International Conference on Digital Interactive Media Entertainment and Arts (DIME-ARTS 2006) (Oct 2006)
10. Park, H., Kim, K.J.: Learning to play fighting game using massive play data. In: Computational Intelligence and Games (CIG), 2014 IEEE Conference on. pp. 1–2 (Aug 2014)
11. Ponsen, M., Spronck, P., Muñoz Avila, H., Aha, D.W.: Knowledge Acquisition for Adaptive Game AI. Sci. Comput. Program. 67(1), 59–75 (Jun 2007)
12. Ricciardi, A., Thill, P.: Adaptive AI for fighting games. http://cs229.stanford.edu/proj2008/RicciardiThill-AdaptiveAIForFightingGames.pdf (Dec 2008), accessed: 2015-02-02
13. Saini, S., Dawson, C., Chung, P.: Mimicking player strategies in fighting games. In: Games Innovation Conference (IGIC), 2011 IEEE International. pp. 44–47 (Nov 2011)
14. Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., Postma, E.: Adaptive game AI with dynamic scripting. Machine Learning 63(3), 217–248 (2006)
15. Szita, I., Ponsen, M., Spronck, P.: Effective and Diverse Adaptive Game AI. Computational Intelligence and AI in Games, IEEE Transactions on 1(1), 16–27 (March 2009)
16. Thawonmas, R., Osaka, S.: A Method for Online Adaptation of Computer-game AI Rulebase. In: Proceedings of the 2006 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology. ACM (2006)
17. Thunputtarakul, W., Kotrajaras, V.: Data Analysis for Ghost AI Creation in Commercial Fighting Games. In: GAMEON. pp. 37–41 (2007)
18. Timuri, T., Spronck, P., van den Herik, H.J.: Automatic Rule Ordering for Dynamic Scripting. In: Artificial Intelligence and Interactive Digital Entertainment (AIIDE) ,2007 AAAI Conference on. pp. 49–54 (2007)
19. Yamamoto, K., Mizuno, S., Chu, C.Y., Thawonmas, R.: Deduction of fighting-game countermeasures using the k-nearest neighbor algorithm and a game simulator. In: Computational Intelligence and Games (CIG), 2014 IEEE Conference on. pp. 1–5 (Aug 2014)